**Proceedings Paper:**
Otoni, R., Cavalcanti, A. L. C. orcid.org/0000-0002-0831-1976 and Sampaio, A. C. A. (2017) Local Analysis of Determinism for CSP. In: Cavalheiro, S. and Fiadeiro, J., (eds.) Formal Methods:Foundations and Applications : 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 — December 1, 2017, Proceedings. Lecture Notes in Computer Science . Springer , pp. 107-124.

https://doi.org/10.1007/978-3-319-70848-5_8

# Local Analysis of Determinism for CSP

Rodrigo Otoni[1] ✉, Ana Cavalcanti[2], and Augusto Sampaio[1]

[1] Universidade Federal de Pernambuco, Centro de Informática, Brazil,
{rbo2,acas}@cin.ufpe.br
[2] University of York, Department of Computer Science, UK,
ana.cavalcanti@york.ac.uk

**Abstract.** Nondeterminism is an inevitable constituent of any theory that describes concurrency. For the validation and verification of concurrent systems, it is essential to investigate the presence or absence of nondeterminism, just as much as deadlock or livelock. CSP is a well established process algebra; the main tool for practical use of CSP, the model checker FDR, checks determinism using a global analysis. We propose a local analysis, in order to improve performance and scalability. Our strategy is to use a compositional approach where we start from basic deterministic processes and check whether any of the composition operators introduce nondeterminism. We present the algorithms used in our strategy and experiments that show the efficiency of our approach.

**Keywords:** Model Checking. FDR. Performance. Experiments.

## 1 Introduction

Deadlock, livelock, and nondeterminism analyses are crucial in the specification and design of concurrent systems. Nondeterminism is expected in abstract models, but may indicate problems in concrete designs. Verification techniques to investigate the presence or absence of all these properties in a model are essential for validation and verification of concurrent systems.

Deadlock and livelock have been investigated in depth, and there are very efficient tools available [1–5, 8]. Determinism has been less studied. It is, however, specially important in notations for refinement, where nondeterminism is used for abstraction. It is an inevitable constituent of any theory that describes concurrency where some form of arbitration is present [9].

CSP is a well established process algebra that is accompanied by a set of robust tools that allow its practical use both in academia and in industry. In particular, CSP is capable of modelling both explicit and implicit nondeterminism, such as the ones that can be introduced by parallelism, internal communications, or renaming. Its versatility in modelling nondeterminism together with its tool support makes CSP ideal for the analysis of determinism.

FDR [5] is the main tool for practical use of CSP; it is a model checker that takes as input specifications in $\text{CSP}_M$, a machine readable version of CSP. Other tools for CSP (or CSP dialects) like ProB [7] and PAT [11] also implement

analysis strategies for these classical properties. The approach taken by all these tools for checking determinism is, however, based on global analysis, where the entire model is expanded and exhaustively checked. Here, we propose a local analysis strategy for determinism, to improve performance and scalability.

Local analysis has been adopted in verification of deadlock [1–3] and livelock [4]. Here, we present a local strategy for the verification of determinism in models written using a subset of CSP that includes most of its basic operators, with some restrictions on how they can be used in compositions. As far as we know, this is the first approach to local analysis of determinism, not only in the context of CSP, but also of any other formal or semi-formal modelling notation (such as UML), as well as concurrent programming languages.

Next we present CSP and how it defines determinism. In Section 3 we present our strategy. Our experiments and their results are discussed in Section 4. Finally we present our final remarks and future work in Section 5.

## 2 Background

We present here the background material to our work: CSP in Section 2.1 and its notion of determinism in Section 2.2.

### 2.1 CSP

CSP is a process algebra that can be used to describe systems as interacting components. These components, called processes, are independent entities that interact among themselves and with the environment. The interactions, called events, are atomic, instantaneous, and synchronous messages. The main CSP constructs are presented below; further information can be found in [6, 9].

CSP has two basic processes, $SKIP$ and $STOP$; the former does nothing and terminates, and the latter deadlocks. A prefixing $a \rightarrow P$ is initially capable of performing the event $a$ and then behaves like the process $P$. Events can be compound to communicate data. For instance, $c.5$ is the event that represents the transmission of the value 5 through the channel $c$.

Guards and conditionals are used in processes $g \ \& \ P$ and if $b$ then $P$ else $Q$. The former behaves as $P$ if $g$ is true, and as $STOP$ otherwise. The latter behaves as $P$ if $b$ is true, and as $Q$ otherwise. Sequential composition is written as $P \,;\, Q$, which behaves as $P$, until it finishes, and then behaves as $Q$.

The process $P \ \Box \ Q$ is the external choice between $P$ and $Q$, resolved in favour of either of them when the environment agrees on their initials, the sets of events that they initially offers. In the internal choice, $P \sqcap Q$, the environment has no control over how the choice is resolved, which is nondeterministic. To make events internal to the process $P$ we can write $P \setminus X$, which hides the events in the set $X$ from the environment.

To model parallelism in CSP we have various options. The process $P \ ||| \ Q$ is the interleaving of $P$ and $Q$; in this composition $P$ and $Q$ behave independently. Another composition is the generalised parallelism, $P[\![X]\!]Q$, in which $P$ and $Q$

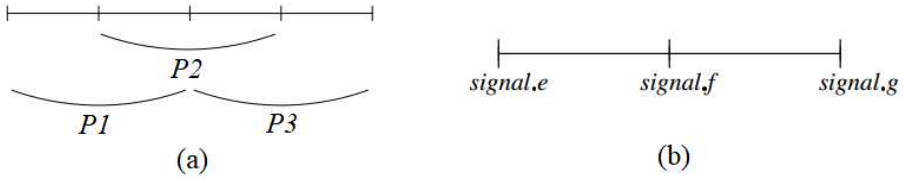


**Fig. 1.** Three overlapping pairs of segments (a), and signals of a pair of segments (b); modified from [10].

synchronise on the events in the set $X$, but allow the events outside of $X$ to occur independently; if $X$ is the empty set, the operator behaves as interleaving.

As an example, we present a specification of a railway network from [10]. It is composed by a series of segments of tracks, with a signal between every two adjacent segments used to control the flow of trains. The segments are organised in overlapping pairs, as shown in Figure 1(a), with segment pairs $P_1$, $P_2$ and $P_3$. In its initial state the railway can have a number of trains in specific segments. A safety requirement is that no two trains should be in adjacent segments.

Each pair of segments has three signals: $e$, which indicates a train entering the pair; $f$, which indicates the train moving from the first to the second segment; and $g$, which indicates the train leaving the pair. A pair of segments is modelled as a process that can communicate three events, $signal.e$, $signal.f$, and $signal.g$, corresponding to the signals $e$, $f$, and $g$. A graphical representation of a pair of segments can be seen in Figure 1(b). To deal with all possible initial states of a pair, three processes are defined. *Pair_Empty* specifies a pair that is initially empty, *Pair_First*, a pair in which a train is initially in its first segment, and *Pair_Second*, a pair in which a train is initially in its second segment.

$$Pair\_Empty = signal.e \rightarrow signal.f \rightarrow signal.g \rightarrow Pair\_Empty$$
$$Pair\_First = signal.f \rightarrow signal.g \rightarrow signal.e \rightarrow Pair\_First$$
$$Pair\_Second = signal.g \rightarrow signal.e \rightarrow signal.f \rightarrow Pair\_Second$$

To model a network we compose a number of instances of pairs of segments in parallel, with each instance having its own signals defined according to its position in the network. In Figure 2 we present an example of a cyclic network that has four pairs and four segments, with the last and the first segments being adjacent to each other. Figure 2(a) gives an overview of the complete network, in which there is initially a single train in the segment demarcated by signal.0 and signal.1. Figure 2(b) shows the four segment pairs (from $Pair0$ to $Pair3$).

The CSP processes that describe the four segment pairs are presented in Figure 3. Note that these processes define the initial state of each segment pair. Therefore, although the $Pair0$ segment pair is formed of the two segments demarcated by $signal.0$ and $signal.1$, and $signal.1$ and $signal.2$, the process is written as $Pair0 = signal.1 \rightarrow signal.2 \rightarrow signal.0 \rightarrow Pair0$, because the train is in the first segment of this pair, and the next relevant event it must communicate

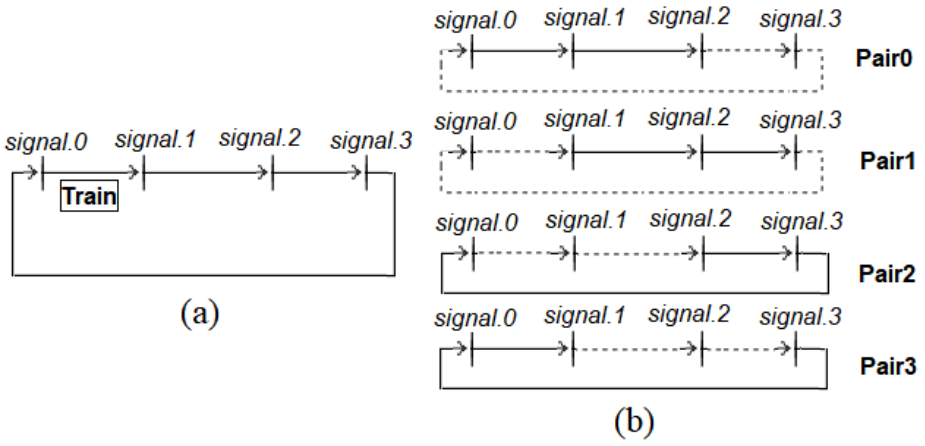


**Fig. 2.** Graphical representation of a network (a), and its pairs (b).

is $signal.1$, indicating the train moving from the first to the second segment of $Pair0$. So $Pair0$ follows the form of $Pair\_First$, previously explained. Similarly, $Pair1$ and $Pair2$ are modelled as $Pair\_Empty$, since the train is not in any of their segments. Finally, $Pair3$ is modelled as $Pair\_Second$, as the train is in the second segment of this pair. The composition of the pairs is made using the generalised parallel operator, since the signals of a pair need to synchronise with the signals of its adjacent pairs.

$$
\begin{aligned}
&Pair0 = signal.1 \rightarrow signal.2 \rightarrow signal.0 \rightarrow Pair0\\
&Pair1 = signal.1 \rightarrow signal.2 \rightarrow signal.3 \rightarrow Pair1\\
&Pair2 = signal.2 \rightarrow signal.3 \rightarrow signal.0 \rightarrow Pair2\\
&Pair3 = signal.1 \rightarrow signal.3 \rightarrow signal.0 \rightarrow Pair3\\
&SyncSet1 = \{signal.1, signal.2\}\\
&SyncSet2 = \{signal.0, signal.2, signal.3\}\\
&SyncSet3 = \{signal.0, signal.1, signal.3\}\\
&RailwayNetwork = ((Pair0 [\![ SyncSet1 ]\!] Pair1) [\![ SyncSet2 ]\!] Pair2) [\![ SyncSet3 ]\!] Pair3
\end{aligned}
$$

**Fig. 3.** CSP model of the network in Figure 2

The process $RailwayNetwork$ can initially communicate $signal.1$, and afterwards $signal.2$, $signal.3$, $signal.0$, $signal.1$, $signal.2$ and so on. Each pair synchronises its first two signals with the pair on its left and its last two signals with the pair on its right, so when the network communicates $signal.1$, it means that a train is, simultaneously: moving from segment 1 to segment 2 of $Pair0$, entering segment 1 of Pair 1, and leaving $Pair3$.

### 2.2 Semantic Models and Determinism

A deterministic system can be thought of as one that always produces the same output, given a fixed input. CSP has different definitions for this property, depending on the semantic model being used. There are three well established semantic models for CSP: traces, failures, and failures-divergences.

In the traces model, a process $P$ is represented by $traces(P)$, which is the set that contains all sequences of events that $P$ can engage. This model does not allow us to determine if the process is deterministic or not.

In the failures model, a process $P$ is represented by the pair ($traces(P)$, $failures(P)$), with $failures(P)$ being a set of pairs $(s, X)$, where $s$ is a trace of $P$ and $X$ is a set of events that $P$ can refuse after performing $s$. This model captures not only how a process can behave, but also how it cannot behave. The definition of determinism in the failures model is presented below.

**Definition 1 (Determinism in the failures model)** *Process $P$ is deterministic if,* $\forall\, tr : traces(P), a : \Sigma \bullet \neg(tr \frown \langle a \rangle \in traces(P) \land (tr, \{a\}) \in failures(P))$

This definition captures the essence of determinism: a process cannot have the possibility of both accepting and refusing an event at any given state, which can lead to different observable behaviours given the same input.

**Example 1** The process $Ex1a = Pair1 \,\Box\, Pair2$ is deterministic, since $Pair1$ and $Pair2$ are deterministic and the intersection of their initials is empty. Without initial events in common, the environment has a clear choice between $Pair1$ and $Pair2$, which, with their traces and failures, do not violate Definition 1.

The process $Ex1b = Pair1 \,\Box\, Pair3$, on the other hand, is nondeterministic, because $signal.1$ is in the initials of both $Pair1$ and $Pair3$, so, by performing $signal.1$, the environment has no control over how the external choice is resolved, allowing $Ex1b$ to both accept or refuse $signal.2$ afterwards, depending on whether $Pair1$ or $Pair3$ is chosen; the trace $\langle signal.1, signal.2 \rangle$ and the failure ($\langle signal.1 \rangle$,$\{signal.2\}$), for instance, break the condition of Definition 1. □

**Example 2** The composition in $Ex2 = Pair1 \,|||\, Pair2$ is nondeterminitic because we have an event, $signal.3$, after which $Pair1$ and $Pair2$ behave differently. In terms of Definition 1, we note that $\langle signal.1, signal.2, signal.3, signal.1 \rangle$ is a trace of $Ex2$ and ($\langle signal.1, signal.2, signal.3 \rangle$,$\{signal.1\}$) is a failure of $Ex2$. □

Nondeterminism can also arise from divergence, that is captured by the failures-divergences model [6, 9]. Since there are tools that verify divergence in a compositional way [4], our strategy is based on determinism in the failures model.

## 3 Strategy for Local Analysis of Determinism

Our analysis of a process is compositional. If the possibility of nondeterminism is found, the analysis stops and indicates the nondeterministic component.

Our strategy is sound for the verification of determinism, but not complete. When nondeterminism is indicated, we may have found a source of nondeterminism or it may be an inconclusive result. Local approaches to the analysis of classical concurrency properties tend to give up completeness in favour of efficiency gains; see [1–3], for deadlock analysis, and [4], for livelock analysis.

**Example 3** We consider the following processes.

$$Ex3a = a \rightarrow b \rightarrow Ex3a \qquad Ex3c = Ex3a \sqcap Ex3b$$
$$Ex3b = c \rightarrow d \rightarrow SKIP \qquad Ex3d = Ex3a [\![ \{a\} ]\!] Ex3c$$

The process $Ex3c$ is nondeterministic, due to its internal choice. The process $Ex3d$, on the other hand, is deterministic. When analysing $Ex3d$, however, our strategy indicates the nondeterminism in $Ex3c$ and stops. □

In Section 3.1 we present the subset of CSP that our strategy can currently handle, and the metadata gathered for the component processes. In Section 3.2, the rules to check for determinism are presented.

### 3.1 Process Structure and Metadata

In our strategy we deal with two categories of processes, Basic Processes and Composite Processes, defined in Figure 4. Event, Condition, ProcessName, and SetOfEvents are the syntactic categories of the possible events, logical conditions, names of processes, and sets of events of CSP. We assume that all processes are divergence free and do not have parameters.

Due to the nature of the set of operators that can be used to create Basic Processes, they are deterministic by definition. A Composite Process is the result of a composition of Basic Processes or other Composite Processes.

The subset of CSP that we deal with, as can be seen in Figure 4, includes most of the basic operators of CSP. They are, however, restricted on their use. Prefixing, guards, conditionals, and sequential composition can only be used in

Process ::= BasicProcess | CompositeProcess

BasicProcess ::= Event " → " BasicProcess
| Condition " & " BasicProcess
| " if " Condition " then " BasicProcess " else " BasicProcess
| BasicProcess " ; " BasicProcess
| "$SKIP$" | "$STOP$" | ProcessName

CompositeProcess ::=
ProcessName " □ " ProcessName | ProcessName " ⊓ " ProcessName
| ProcessName " ||| " ProcessName | ProcessName "[[" SetOfEvents "]]" ProcessName
| ProcessName " \ " SetOfEvents

**Fig. 4.** BNF of the subset of CSP considered.

the definition of Basic Processes, while external and internal choice, interleaving, generalized parallel, and hiding are restricted to Composite Processes.

Each process in our strategy, upon being verified to be deterministic, is associated with a Set of Possible Behaviours (SPB). This is a set of sets of pairs, with each set of pairs in an SPB representing an alternative behaviour of the process, and each pair in a set representing a parallel behaviour. The first component of each pair in the set is a sequence that represents part of the syntactic structure of the process, and its second component is a set, which stores data relative to synchronisations among components of the process.

**Example 4** The processes $Ex3a$ and $Ex3b$, from Example 3, have the SPBs: SPB($Ex3a$) = $\{\{(\langle a, b, Ex3a\rangle, \varnothing)\}\}$, and SPB($Ex3b$) = $\{\{(\langle c, d, SKIP\rangle, \varnothing)\}\}$. Each one has one set, because they do not have choices, with one pair, which holds the structure of the process, since there is no parallelism; the second element of the pairs is the empty set, also because we do not have any synchronisations between processes. For the process $Ex4 = Ex3a \,\Box\, Ex3b$ we have that SPB($Ex4$) = $\{\{(\langle a, b, Ex3a\rangle, \varnothing)\}, \{(\langle c, d, SKIP\rangle, \varnothing)\}\}$, which captures its two alternative behaviours, that depend on how the choice is resolved. □

Each element of a synchronisation set is itself a pair, with an integer value as the first component, and a set of events as the second component; the events in the set are the ones being synchronised. The integer values identify the sets that match in a synchronisation.

**Example 5** We consider the following processes.

$$Ex5a = a \rightarrow STOP \qquad Ex5c = b \rightarrow Ex5c$$
$$Ex5b = Ex5a [\![\{a\}]\!] Ex3a \qquad Ex5d = Ex5b [\![\{b\}]\!] Ex5c$$

With the generalised parallel operator we add the synchronisation set to all pairs of the SPB of the composition, so, for the process $Ex5b$, we have that SPB($Ex5b$) = $\{\{(\langle a, STOP\rangle, \{(1, \{a\})\}), (\langle a, b, Ex3a\rangle, \{(-1, \{a\})\})\}\}$; the module of the integer value uniquely identifies the synchronisation and its signal is used to differentiate between the two argument processes of the parallelism.

If $Ex5b$ is used in a composition with generalised parallel, we add the new synchronisation set to its pairs. For $Ex5d$, we have the set shown below.

$$SPB(Ex5d) = \left\{ \left\{ \begin{array}{l} (\langle a, STOP\rangle, \{(1, \{a\}), (2, \{b\})\}), \\ (\langle a, b, Ex3a\rangle, \{(-1, \{a\}), (2, \{b\})\}), \\ (\langle b, Ex5c\rangle, \{(-2, \{b\})\}) \end{array} \right\} \right\}$$

When more than one pair has a synchronisation set with the same integer, in this case 2, only one of those pairs need to synchronise with a counterpart with the opposite integer, in this case -2. □

Now we present the formal definition of SPB. It is important to record whether a sequence leads to a recursion or not. To this end, we add the name of the

process that represents the final behaviour of the sequence as its last element. We call the set that contains these new sequences Valid Sequences (VS). With VS it is possible to know if a Basic Process, which can only have one sequence, is cyclic or not just by checking if the last element of its sequence is a process name (indicating a recursion), or if it is $SKIP$ or $STOP$.

**Definition 2 (Valid Sequences (VS))**

$$VS = \{a : \Sigma^*, b : N \cup \{SKIP, STOP\} \bullet a^\frown\langle b\rangle\}$$

*where $N$ is the set that contains all the valid names of processes.*

To record a trace of behaviour in a parallel process with synchronisations, we use Synchronisation Sets, which is the set that contains integers associated with all possible sets of events on which a process can synchronise.

**Definition 3 (Syncronisation Sets (SyncSets))** $SyncSets = \mathbb{P}(\mathbb{Z} \times \mathbb{P}(\Sigma))$

Finally we define the *eTraces* (a shorthand for Enhanced Traces) of a given process $P$. Its elements are pairs whose first element is a $VS$ sequence and the second one is a set of *SyncSets*.

**Definition 4 (Enhanced Traces of P (eTraces(P)))**

$$eTraces(P) = \left\{ et : VS \times SyncSets \mid front(first(et)) \in traces(P) \land \left( \exists\, n \in N \cup \{SKIP, STOP\} \bullet \begin{array}{l} P/front(first(et)) \equiv_{\mathrm{F}} n \\ \land \\ last(first(et)) = n \end{array} \right) \right\}$$

*where $P/t$ represents the behaviour of $P$ after it has performed the trace $t$, and $\equiv_{\mathrm{F}}$ indicates equivalence in the failures model* [9].

An element of *eTraces* represents a possible Basic Process. The restrictions in Definition 4 ensure that each sequence leads the process to a recursive behaviour, or to $SKIP$ or $STOP$. For a process $P$, the set SPB($P$) is a subset of $\mathbb{P}$(eTraces($P$)). The *eTraces* pairs in a set of an SPB represent Basic Processes in parallel, and pairs in different sets represent choices.

Now we present how SPB is calculated. For the Basic Processes, we calculate SPB as shown below; $P$ and $Q$ are processes, and $n$ is a process name.

- $SPB(n) = \{\{(\langle n\rangle, \varnothing)\}\}$
- $SPB(a \rightarrow P) = \{setP : SPB(P) \bullet prefixing\ (\!|\ \{a\} \times setP\ |\!)\}$
- $SPB(g\ \&\ P) = SPB(P)$
- $SPB(\text{if } g \text{ then } P \text{ else } Q) = SPB(P)$
- $SPB(P\ ;\ Q) = \{setP : SPB(P)\ ;\ setQ : SPB(Q) \bullet seqComp\ (\!|\ setP \times setQ\ |\!)\}$

- $prefixing(event, eTrace) = (\langle event\rangle^\frown first(eTrace), \varnothing)$

– $seqComp(eTrace1, eTrace2) = \text{if } last(first(eTrace1)) = SKIP$
$\text{then } (front(first(eTrace1)) \frown first(eTrace2), \varnothing)$
$\text{else } eTrace1$

For a process call, we create a sequence with the call. For a prefixing, $a \rightarrow P$, we apply *prefixing* to all pairs formed of the event $a$ and a sequence in a set of the SPB of P; $(\!|...|\!)$ is the relational image operator. The function *prefixing* yields the original sequence with the new event as its head.

We assume that the predicates in guards and conditionals are always true; in those cases we simply keep the SPB of $P$. In the conditional, if the processes $P$ and $Q$ are not equivalent, the strategy returns the possibility of nondeterminism. With this approach, we record behaviours for the processes that may not be actually possible. The addition of behaviours, however, can only lead to nondeterminism, never remove it. So, as already explained, it is possible that we indicate a nondeterminism that does not exist, but a process defined to be deterministic is guaranteed to be so.

For sequential composition we apply *seqComp* to all pairs of SPB($P$) and SPB($Q$), using relational image. This function returns the front of the first sequence appended with the second sequence, if the first one ends in $SKIP$, or the first sequence unmodified otherwise.

**Example 6** The calculation of SPB($Ex3a$) is shown below.

$\text{SPB}(Ex3a) = \{\{(\langle Ex3a\rangle, \varnothing)\}\}$
$\text{SPB}(b \rightarrow Ex3a) = \{\{(\langle b, Ex3a\rangle, \varnothing)\}\}$
$\text{SPB}(a \rightarrow b \rightarrow Ex3a) = \{\{(\langle a, b, Ex3a\rangle, \varnothing)\}\}$
$\text{SPB}(Ex3a) = \text{SPB}(a \rightarrow b \rightarrow Ex3a)$

We differentiate between the process $Ex3a$ and its recursive call. For sequential composition, $\text{SPB}(Ex3a\,;Ex3b) = \text{SPB}(Ex3a)$, since the sequence of $Ex3a$ ends in a recursion, and $\text{SPB}(Ex3b\,;Ex3a) = \{\{(\langle c, d, a, b, Ex3a\rangle, \varnothing)\}\}$, because the sequence of $Ex3b$ ends in $SKIP$. □

We now present the SPB for the Composite Processes; $X$ is a set of events, and $i$ is a fresh integer, different from zero.

– $SPB(P \,\square\, Q) = SPB(P) \cup SPB(Q)$
– $SPB(P \sqcap Q) = SPB(P)$
– $SPB(P \;|||\; Q) = \{setP : SPB(P)\,;\, setQ : SPB(Q) \bullet setP \cup setQ\}$
– $SPB(P[\![X]\!]Q) = \left\{ \begin{array}{l} setP : SPB(P)\,;\, setQ : SPB(Q) \bullet \\ addSync\,(\!|\, setP \times \{X\} \times \{i\} \,|\!) \\ \cup \\ addSync\,(\!|\, setQ \times \{X\} \times \{-i\} \,|\!) \end{array} \right\}$
– $SPB(P \setminus X) = \{setP : SPB(P) \bullet remove\,(\!|\, setP \times \{X\} \,|\!)\}$

– $addSync(eTrace, X, id) = (first(eTrace), second(eTrace) \cup \{(id, X)\})$
– $remove((T, S), X) = (removeT(T, X), removeS(S, X))$

– $removeT(\langle\rangle, X) = \langle\rangle$
  $removeT(\langle a\rangle^\frown t, X) = \text{if } a \in X \text{ then } remove(t, X) \text{ else } \langle a\rangle^\frown remove(t, X)$
– $removeS(\varnothing, X) = \varnothing$
  $removeS(\{(id, evSet)\} \cup s, X) = \{(id, evSet \setminus X)\} \cup removeS(s, X)$

For external choice, we get the union of the sets of the operands. For internal choice, since the composition is only deterministic if both operands are equivalent, we simply keep the SPB of one of them. For interleaving, for every pair of sets of SPB($P$) and SPB($Q$), we record their union. The calculation for a generalised parallel is similar to that of an interleaving, but we also add the new synchronisation to the elements of the sets, using the function *addSync*. For hiding we remove the elements in $X$ from SPB($P$), with the function *remove*.

**Example 7** Considering the processes $Ex7a = a \rightarrow b \rightarrow c \rightarrow Ex7a$, and $Ex7b = Ex7a [\![\{b\}]\!] Ex7a$, we calculate SPB($Ex7b \setminus \{b\}$).

SPB($Ex7b$) = {{($\langle a, b, c, Ex7a\rangle$,{(1,{b})}), ($\langle a, b, c, Ex7a\rangle$,{(-1,{b})})}}
SPB($Ex7b \setminus \{b\}$) = {{($\langle a, c, Ex7a\rangle$,{(1,{})}), ($\langle a, c, Ex7a\rangle$,{(-1,{})})}}

□

If a synchronisation introduces deadlock or if a synchronisation channel is hidden, there is the possibility that our strategy considers invalid behaviours of the process. This, however, can only introduce nondeterminism, never remove it.

**Example 8** We consider the following processes.

$$Ex8a = b \rightarrow a \rightarrow c \rightarrow d \rightarrow Ex8a \qquad Ex8b = Ex7a [\![\{a, b\}]\!] Ex8a$$

The process $Ex8b$ is deterministic, because it is deadlocked from the start. Our strategy, however, predicts a nondeterministic behaviour when both $Ex7a$ and $Ex8a$ offer event $c$ to the environment, which never happens. □

Pairs of a set of a SPB are equivalent, $\equiv$, if the front of their sequences are equal, both either recurse or end in $SKIP$ or $STOP$, and they have the same meaningful synchronisations with equivalent pairs. A synchronisation is meaningful if it involves at least one of the events in the sequence of the pair.

**Definition 5 (Meaningful Synchronisations)** *Given a pair (Seq, SetOfSyncs), a synchronisation set sync ∈ SetOfSyncs is meaningful if sync ∩ ran(Seq) ≠ ∅.*

**Example 9** We consider the following SPBs.

SPB($Ex9a$) = {{($\langle a, b, c, P\rangle$,{(1,{r})}), ($\langle x, y, SKIP\rangle$,{(-1,{r})})}}
SPB($Ex9b$) = {{($\langle a, b, c, P\rangle$,{(2,{r})}), ($\langle x, y, SKIP\rangle$,{(-2,{r})})}}
SPB($Ex9c$) = {{($\langle a, b, c, Q\rangle$,∅), ($\langle x, y, SKIP\rangle$,∅)}}

They are all equivalent, since the only difference between them is their synchronisation sets, with the synchronisations of $Ex9a$ and $Ex9b$ not being meaningful, as they do not affect the sequences, and $Ex9c$ not having synchronisations. □

In the next section, we present the algorithms that use the SPB of component processes to check determinism of a composite process.

**Algorithm 1** External Choice (P,Q)

```
1: for each setP ∈ SPB(P), setQ ∈ SPB(Q) do
2:    for each elemP ∈ setP, elemQ ∈ setQ do
3:       if head(first(elemP)) == head(first(elemQ)) ∧ ¬(setP ≡ setQ) then
4:          return false
5: return true
```

**Fig. 5.** Algorithm to check if external choice introduces nondeterminism.

### 3.2 Composition Rules

The algorithms that verify if the compositions are deterministic return true if the given composition is deterministic, and false otherwise. We present here the algorithms for external choice, and parallelism. The algorithms for internal choice and hiding can be found in the extended version of this paper[1].

#### External Choice

The external choice, with our restrictions, can only introduce nondeterminism if its two operands have at least one common initial event, since these are their only points of interaction. In this scenario the composition is deterministic only if the two processes have the same behaviour after every common initial event. The algorithm for this operator can be seen in Figure 5. It checks for all pairs of sets of SPB($P$) and SPB($Q$) if they have sequences that start with the same event. If they do, those sets need to be equivalent not to introduce nondeterminism.

#### Internal Choice

An internal choice only results in a deterministic process if its operands have the same behaviour. The algorithm for this operator simply checks if the SPBs of both operands are equivalent, that is, if the SPBs have equivalent sets.

#### Parallelism

We deal with parallelism in two forms: interleaving and generalised parallel. We first discuss how interleaving can introduce nondeterminism. Afterwards, we present our considerations about generalised parallel. Finally, we show the algorithm for the verification of parallel compositions.

Differently from external and internal choice, with interleaving, as well as with the other parallel operators, both operands execute at the same time, so we must take into account all of their events, not only the initials. With interleaving,

[1] http://www.cin.ufpe.br/∼rbo2/SBMF2017.zip

we need to consider that when one of its operands is offering a specific event to the environment, the other operand can be offering any of its events.

The condition for a composition using interleaving to be deterministic is that, after each event in common to both processes, the composition needs to offer the same events to the environment, no matter which process performs the event, so the environment does not observe any different behaviour.

**Example 10** We consider the following processes.

$$Ex10a = a \rightarrow b \rightarrow Ex10a \qquad Ex10d = Ex10a \;|||\; Ex10b$$
$$Ex10b = a \rightarrow Ex10b \qquad Ex10e = Ex10b \;|||\; Ex10c$$
$$Ex10c = b \rightarrow Ex10c \qquad Ex10f = Ex10a \;|||\; Ex10e$$

The process $Ex10d$ is nondeterministic, because after performing event $a$, the environment can synchronise on either $a$ again or on $a$ and $b$, depending if $a$ was performed by $Ex10a$ or $Ex10b$. The process $Ex10e$ is deterministic, because the alphabets of its components are disjoint, so there are no events in common. The composition in $Ex10f$ is deterministic, because, although there is an intersection of the alphabets, events $a$ and $b$ are always available to the environment. □

Generalised parallel allows us to have parallelism with synchronisations. The events that are not in the synchronisation set are analysed in a similar way to what is done with interleaving, and the events in the synchronisation set cannot introduce nondeterminism on their own, because each synchronised event happens only once and both operands engage in this event.

**Example 11** The process $Ex11 = Ex10a[\![\{a\}]\!]Ex10b$, differently from $Ex10d$, is deterministic, because the event $a$ is in the synchronisation set, so, after it occurs, the only possibility for the parallel composition is to offer event $b$. □

We use the same algorithm for the two forms of parallelism discussed. It receives the processes being composed and the synchronisation set. For interleaving, the synchronisation set is empty. The algorithm is presented Figure 6.

Algorithm 2 iterates over all pairs of behaviours of $P$ and $Q$, evaluating all scenarios. In each iteration, it initially defines *avEvents* (line 2), the set of events that is always available to the environment, in the given pair of behaviours.

To calculate *avEvents* we use *setOfAvailableEvents*, which yields a set of events that must belong to an Enhanced Trace that has only one event in its sequence and is recursive. These sequences stand out because they do not lead to a change in the state of the composition. Another requirement is that these events need to be able to occur freely, which can be denied by synchronisations.

**Example 12** We consider the following SPBs.

$$\text{SPB}(Ex12a) = \{\{(\langle a, b, P\rangle,\varnothing), (\langle c, Q\rangle,\varnothing), (\langle d, SKIP\rangle,\varnothing), (\langle e, R\rangle,\varnothing)\}\}$$
$$\text{SPB}(Ex12b) = \{\{(\langle x, S\rangle,\{(1,\{x\})\}), (\langle x, y, T\rangle,\{(-1,\{x\})\}), (\langle e, f, U\rangle,\varnothing)\}\}$$

**Algorithm 2** Parallelism (P,Q,X)

```
1: for each setP ∈ SPB(P), setQ ∈ SPB(Q) do
2:    avEvents = setOfAvailableEvents(setP,setQ,X)
3:    for each elemP ∈ setP, elemQ ∈ setQ do
4:       if head(first(elemP)) == head(first(elemQ)) ∧ head(first(elemP)) ∉ X ∧
         (¬allSetsEquiv(SPB(P)) ∨ ¬allSetsEquiv(SPB(Q))) then
5:          return false
6:       for each evP ∈ front(first(elemP)), evQ ∈ front(first(elemQ)) do
7:          if evP == evQ ∧ evP ∉ X then
8:             eventsP = avEvents ∪ nextEvents(elemP,evP,setP) ∪ {evQ}
9:             eventsQ = avEvents ∪ nextEvents(elemQ,evQ,setQ) ∪ {evP}
10:            if eventsP ≠ eventsQ then
11:               return false
12: return true
```

**Fig. 6.** Algorithm to check if parallelism introduces nondeterminism.

If we execute Parallelism($Ex12a$,$Ex12b$,{e}), we have one iteration of the algorithm with $avEvents = \{c\}$. The pairs with the sequences $\langle a, b, P\rangle$, $\langle x, y, T\rangle$, and $\langle e, f, U\rangle$ are discarded for having more than one event. The pair with $\langle d, SKIP\rangle$ is discarded for not being recursive. The pairs with sequences $\langle x, S\rangle$, and $\langle e, R\rangle$ are discarded due to their synchronisations, the former with the pair ($\langle x, y, T\rangle$,{(-1,{$x$})}, and the latter with the synchronisation being introduced in this composition, through the synchronisation set {e}. □

With *avEvents* calculated, Algorithm 2 starts checking each pair of elements of the behaviour of $P$ and $Q$. If we have two elements with the same initial events (line 4), then, for each operand, all the sets in its SPB need to be equivalent, which is checked by *allSetsEquiv*.

**Example 13** We consider the following processes.

$$Ex13a = Ex10a \,\Box\, Ex10c \qquad\qquad Ex13b = Ex13a \,|||\, Ex10c$$

The composition $Ex13b$ is nondeterministic, because, after performing $b$, the environment does not know if $a$ is still available, since $b$ can be performed by $Ex13a$ or $Ex9c$, so it is possible to accept or refuse $a$, given the circumstances. Algorithm 2 returns false because the conditional in line 4 returns true, having $allSetsEquiv(SPB(Ex13a)) = false$. □

Requiring that the SPB of each process have all sets equal is, however, not enough to ensure determinism, as we can see in the next example.

**Example 14** We consider the following processes.

$$Ex14a = a \rightarrow b \rightarrow a \rightarrow Ex14a \qquad\qquad Ex14b = Ex14a \,|||\, Ex14a$$

The process $Ex14b$ is nondeterministic, because after the environment performs the trace $\langle a, b, a\rangle$ it is possible to accept or refuse $b$. During the evaluation, the conditional in line 4 returns true, $allSetsEquiv(SPB(Ex14a)) = true$, so nondeterminism is not identified by Algorithm 2 at this point. □

We then have the core of Algorithm 2 (line 6). The algorithm checks, for each event $e$ that is in the intersection of the alphabets, if the events available to the environment, after $e$ in $P$ is performed, is equal to the events available after $e$ in $Q$ is performed. If they are not, then a source of nondeterminism has been found. This verification is only carried out if $e$ is not in the synchronisation set.

We assume that the two sequences that we are analysing at a given moment are offering specific events, but we do not assume anything of the other sequences. The events available to the environment after the execution of the event in each process are given by the union of three sets: *avEvents*; the set of events that are available in the process that performed the event, after its execution; and the set that contains the event in question, that is still available in the other process.

The function *nextEvents* (lines 8 and 9) returns the set of events that a process offers to the environment after one of its events, $e$, has occurred; in Algorithm 2, $e$ can be $evP$ or $evQ$. First it adds the event that comes after $e$, if any, to the return set. Afterwards it checks if the pair that contains $e$ synchronises with other pairs, and if $e$ is present in those pairs, which leads to a change in their states as well. If $e$ is indeed present, then the events after it in these events will also be included in the return set.

**Example 15** For process $Ex14b$, we have $avEvents = \varnothing$. The conditional in line 7 returns true for the first events in both $Ex14a$ operands, with $eventsP$ and $eventsQ$ being both the result of $\varnothing \cup \{b\} \cup \{a\}$, so $eventsP == eventsQ$ (line 10). For the first event of the first operand, $a$, and the second event of the second operand, $b$, the conditional in line 7 returns false. For the first event of the first operand, $a$, and the third event of the second operand, $a$, the conditional in line 7 returns true, but $eventsP = \varnothing \cup \{b\} \cup \{a\}$, and $eventsQ = \varnothing \cup \{a\} \cup \{a\}$, so Algorithm 2 returns false in line 11. □

**Example 16** We consider the following processes.

$$Ex16a = a \to b \to c \to Ex16a \qquad Ex16d = Ex16a \;|||\; Ex16b$$
$$Ex16b = d \to e \to f \to Ex16b \qquad Ex16e = Ex16c [\![ \{d\} ]\!] Ex16d$$
$$Ex16c = d \to e \to g \to Ex16c$$

To check if $Ex16e$ is deterministic we use $\mathrm{SPB}(Ex16c) = \{\{(\langle d, e, g, Ex16c\rangle, \varnothing)\}\}$, and $\mathrm{SPB}(Ex16d) = \{\{(\langle a, b, c, Ex16a\rangle, \varnothing), (\langle d, e, f, Ex16b\rangle, \varnothing)\}\}$. There is one set in $\mathrm{SPB}(Ex16a)$ and in $\mathrm{SPB}(Ex16b)$, so Algorithm 2 performs one iteration, with $avEvents = \varnothing$, since there is no sequence with a freely occurring event.

We check every pair being analysed (line 3). For the pairs $(\langle d, e, g, Ex16c\rangle, \varnothing)$ and $(\langle a, b, c, Ex16a\rangle, \varnothing)$, the conditional in line 4 returns false, since the head of the sequences is different, and so does all six occurrences of the conditional in line

7, inside the loop in line 6, because the two sequences have no event in common. For $(\langle d, e, g, Ex16c\rangle, \varnothing)$ and $(\langle d, e, f, Ex16b\rangle, \varnothing)$, the conditional in line 4 also returns false, this time because $d \in X$, but when the loop in line 6 executes for the second event of each sequence, the conditional in line 7 returns true. In this case we have that $eventsP = \varnothing \cup \{g\} \cup \{e\}$ and $eventsQ = \varnothing \cup \{f\} \cup \{e\}$, so Algorithm 2 returns false. □

**Example 17** We consider the following processes.

$$Ex17a = Ex16a \,\Box\, Ex16b \qquad\qquad Ex17b = Ex17a [\![\{e\}]\!] Ex16c$$

We have $\mathrm{SPB}(Ex17a) = \{\{(\langle a, b, c, Ex16a\rangle, \varnothing)\},\ \{(\langle d, e, f, Ex16b\rangle, \varnothing)\}\}$. The application of Algorithm 2 to $Ex17b$ occurs similarly to that of $Ex16e$, but two iterations occur. The first iteration, with the sets $\{(\langle a, b, c, Ex16a\rangle, \varnothing)\}$, and $\{(\langle d, e, g, Ex16c\rangle, \varnothing)\}$ occurs without problems. The second iteration, with the sets $\{(\langle d, e, f, Ex16b\rangle, \varnothing)\}$ and $\{(\langle d, e, g, Ex16c\rangle, \varnothing)\}$, however, leads the conditional in line 4 to return true when analysing the only two pairs, because $allSetsEquiv(\mathrm{Ex17a}) = \mathrm{false}$, leading Algorithm 2 to return false. □

The way we calculate the available events after an event occurs is the main source of efficiency gain when we deal with parallelism. A global analysis would consider all possible states of the other sequences to carry out the verification. Our strategy, with the use of *avEvents*, considers only a small part of the state space. The sequences that perform various events before a recursion, $SKIP$, or $STOP$, can offer different events, depending of their state, but, more importantly, can refuse to offer them. Since we cannot rely on the events of these sequences to ensure determinism we discard them altogether. For $Ex16d$, for instance, FDR4 visits 54 states, while our strategy only considers 9 states.

We have implemented all the algorithms presented in this section, plus the algorithm to check hiding, to construct a prototype determinism checker. In the next section, we show the results of experiments carried out using this prototype.

## 4 Experimental Results

We performed a number of experiments to compare FDR4 with our prototype. The railway network described in Section 2.1 is our main case study. To evaluate every algorithm, we also considered processes involving external and internal choice, interleaving, and hiding. The prototype and the files used in the experiments are available online, on the link referenced on Page 11.

We used two models of the railway network: the original, deterministic, model, and a modified nondeterministic model, with an error in the last two pairs of tracks. For each model we consider three scenarios, consisting of one, six, and eleven trains, respectively. For each scenario, eight instances are evaluated, with an increasing number of pairs of tracks. The instance number indicates the number of pairs of tracks in it. The results of the experiments with the railway network can be seen in tables 1 to 6; the * indicates an out-of-memory error.

**Table 1.** Deterministic instances with one train in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.12s | 0.35s |
| 50 | 0.22s | 0.46s |
| 75 | 0.32s | 0.50s |
| 100 | 0.37s | 0.56s |
| 500 | 2.94s | 1.38s |
| 1000 | 8.67s | 2.19s |
| 5000 | 4m20.06s | 20.13s |
| 10000 | * | 1m29.63s |

**Table 2.** Nondeterministic instances with one train in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.13s | 0.38s |
| 50 | 0.23s | 0.43s |
| 75 | 0.30s | 0.53s |
| 100 | 0.48s | 0.61s |
| 500 | 3.65s | 1.38s |
| 1000 | 12.76s | 2.21s |
| 5000 | 10m33.92s | 19.74s |
| 10000 | * | 1m26.83s |

**Table 3.** Deterministic instances with six trains in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.52s | 0.35s |
| 50 | 3m6.67s | 0.44s |
| 75 | * | 0.50s |
| 100 | * | 0.60s |
| 500 | * | 1.37s |
| 1000 | * | 2.18s |
| 5000 | * | 19.55s |
| 10000 | * | 1m24.65s |

**Table 4.** Nondeterministic instances with six trains in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.17s | 0.35s |
| 50 | 2.16s | 0.45s |
| 75 | 22.56s | 0.50s |
| 100 | 2m11.05s | 0.55s |
| 500 | * | 1.36s |
| 1000 | * | 2.07s |
| 5000 | * | 19.88s |
| 10000 | * | 1m24.16s |

**Table 5.** Deterministic instances with eleven trains in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.18s | 0.34s |
| 50 | * | 0.43s |
| 75 | * | 0.51s |
| 100 | * | 0.58s |
| 500 | * | 1.37s |
| 1000 | * | 2.20s |
| 5000 | * | 19.97s |
| 10000 | * | 1m22.75s |

**Table 6.** Nondeterministic instances with eleven trains in the railway.

| Instance | FDR4 | Prototype |
|---|---|---|
| 25 | 0.13s | 0.37s |
| 50 | 12.77s | 0.43s |
| 75 | 8m25.50s | 0.51s |
| 100 | * | 0.58s |
| 500 | * | 1.45s |
| 1000 | * | 2.13s |
| 5000 | * | 19.98s |
| 10000 | * | 1m23.92s |

The experiments were run in a server with an Intel Core i7-2600k, 16GB of RAM, 160GB of SSD, and Ubunto 17.94 64-bit. We used FDR 4.2.0. The results show that while for smaller examples FDR has a better performance, due to the overhead of calculation of metadata in our approach, it struggles to analyse large parallel systems. Our prototye was able to analyse the largest instance in less than two minutes, a very promising result.

To analyse the impact of external choice, our experiments consist of a number of processes in the form $Basici = a.i \rightarrow b.i \rightarrow c.i \rightarrow Basici$, all composed with

this operator. Nondeterministic instances were created by modifying the last process to $Basici = a.(i-1) \rightarrow b.i \rightarrow c.i \rightarrow Basici$. For the evaluation of the hiding operator, we modified the instances of the external choice experiments by hiding event $b.i$ after each composition, with the nondeterministic instances hiding $a.i$, instead of $b.i$, after the last composition.

The experiments with internal choice consist of compositions of processes of the form $Basici = a.0 \rightarrow b.0 \rightarrow c.0 \rightarrow Basici$, with the nondeterministic instances having $b.1$, instead of $b.0$, in the last process. For interleaving, we compose processes with the same structure of the processes used in the external choice experiments, with the nondeterministic instances having their last two processes ending with new events, $... \rightarrow c.(i-1) \rightarrow d \rightarrow e \rightarrow f \rightarrow Basic(i-1)$ and $... \rightarrow c.i \rightarrow d \rightarrow e \rightarrow g \rightarrow Basici$.

The results of our additional experiments are in the extended version of this paper. With them we identified that FDR4 does not scale well, specially with interleaving. For all problems considered, except the 10000 instance of our hiding experiment, the prototype completed its analysis in less than two minutes. It is fair to remember that, while FDR4 would correctly identify processes like $Ex3d$ as deterministic, our prototype would indicate the possibility of nondeterminism.

## 5 Conclusion

In this paper we propose a local analysis for the verification of determinism, considering a subset of CSP. We analyse each composition that is part of the process being verified, gathering metadata about them. With the metadata we gather, we are able to guarantee determinism by only checking conditions on the argument processes of the composition. We performed some experiments and the results show that our approach scales better than that of FDR4, the main tool for verification of CSP models, specially when dealing with interleaving.

Local analysis has been used for the verification of properties of concurrency. For livelock, a compositional strategy that handles a subset of CSP similar to our own is presented in [4]. For deadlock, there are works aimed at CSP that involve adherence to deadlock-free patters [2, 3], with a focus on the analysis of cyclic networks of processes. Recent improvements on local deadlock analysis for CSP are reported in [1], but this work also presents an incomplete strategy. As already mentioned, we are not aware of any other approach to compositional analysis of determinism, so our work is an original contribution in this direction.

The composition of techniques that verify deadlock, livelock, and determinism locally is possible. By identifying a subset of CSP shared by all of them, an integrated approach to analyse all the three classical properties is viable.

Our strategy can be improved. We will widen the considered subset of CSP, removing some of the restrictions, to allow non-tail recursion and parameters. We will also prove the correctness of the algorithms and perform more case studies.

## Acknowledgements

This work was partially supported by INES (grants CNPq/465614/2014-0, and FACEPE/APQ/0388-1.03/14) and FACEPE (grant IBPG-0074-1.03/16). We thank Madiel Conserva Filho and Joabe Jesus Júnior for the helpful discussions.

## References

1. Antonino, P.R.G., Gibson-Robinson, T., Roscoe, A.W.: Tighter Reachability Criteria for Deadlock-Freedom Analysis. In: FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. pp. 43–59 (2016)
2. Antonino, P.R.G., Oliveira, M.V.M., Sampaio, A., Kristensen, K.E., Bryans, J.W.: Leadership Election: An Industrial SoS Application of Compositional Deadlock Verification. In: NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings. pp. 31–45 (2014)
3. Antonino, P.R.G., Sampaio, A., Woodcock, J.: A Refinement Based Strategy for Local Deadlock Analysis of Networks of CSP Processes. In: FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. pp. 62–77 (2014)
4. Filho, M.S.C., Oliveira, M.V.M., Sampaio, A., Cavalcanti, A.: Local Livelock Analysis of Component-Based Models. In: Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings. pp. 279–295 (2016)
5. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 - A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference. pp. 187–201 (2014)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
7. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods. pp. 855–874. LNCS 2805, Springer-Verlag (2003)
8. Ramos, R., Sampaio, A., Mota, A.: Systematic Development of Trustworthy Component Systems. In: FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. pp. 140–156 (2009)
9. Roscoe, A.: Understanding Concurrent Systems. Springer-Verlag New York, Inc., New York, NY, USA, 1st edn. (2010)
10. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. John Wiley & Sons, Inc., New York, NY, USA, 1st edn. (1999)
11. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. pp. 709–714. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)